

第3章

μITRON 4.0仕様準拠のマルチプロセッサ対応OSの実装

――「排他制御」に対するカーネルの働きを理解する

本田晋也

ここでは，機能分散型マルチプロセッサに対応したリアルタイムOS「TOPPERS/FDMPカーネル」の実装事例を紹介する．米国Altera社と米国Xilinx社のFPGAのソフト・マクロCPUをターゲットとして開発した．なお，実装にあたっては，μITRON 4.0仕様の標準プロファイルに準拠した「TOPPERS/JSPカーネル」をベースにしている．（編集部）

機能分散型マルチプロセッサ注1用の「TOPPERS/FDMPカーネル（以下，FDMPカーネル）」では，プロセッサごとに独立したオブジェクト・コードが実行されます（**図1**）．すなわち，システムとしては，プロセッサごとに異なるカーネルが協調して動作することになります．

注1：本稿で言う「機能分散型マルチプロセッサ」とは，各プロセッサが別々のソフトウェア基盤（OSなど）を持ち，プロセッサごとの処理が固定されているアーキテクチャを指す．

注2：直接操作法はメモリ・アクセスが高速な場合に有効だが，共有メモリは必須となる．これとは別に「遠隔呼び出し法」というオブジェクト操作方法がある．この方法では共有メモリは不要だが，対象オブジェクトが所属するプロセッサに操作を依頼するため，操作のたびにそのプロセッサに割り込みをかけることになる．

1 実装に際しての前提

FDMPカーネルでは，API（application program interface）の発行によってほかのプロセッサのオブジェクトを操作するとき，そのプロセッサのメモリにある操作対象オブジェクトの管理ブロック（メモリ）を直接操作注2します．操作の結果，対象オブジェクトが所属するプロセッサでディスパッチ（タスク切り替え）が必要となった場合のみ，そのプロセッサに割り込みを発行してディスパッチを要求します．割り込みを受け付けたプロセッサは，ディスパッチ処理を行います（**図2**）．

● カーネルが動作可能なCPUアーキテクチャ

現状のFDMPカーネルが動作するのは，以下に示すアーキテクチャを持つマルチプロセッサ・システムです．

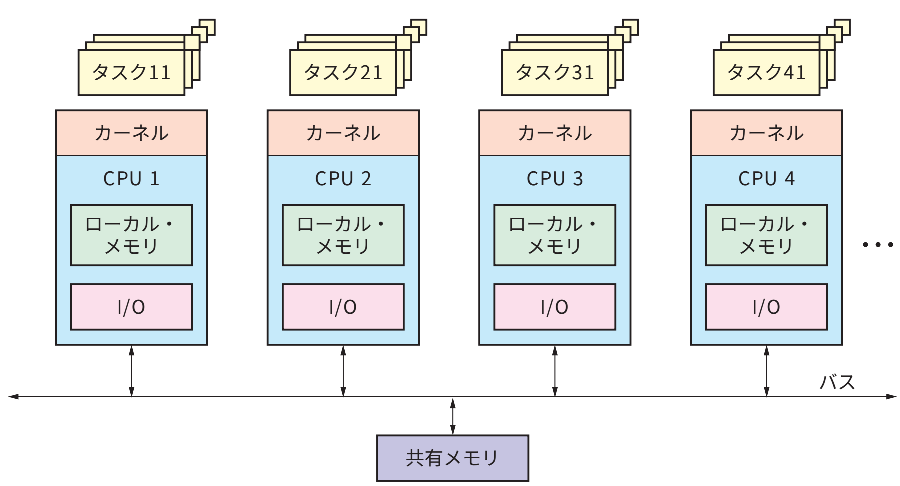


図1 機能分散型マルチプロセッサの基本モデル
FDMPカーネルの対象となるマルチプロセッサの構成．プロセッサごとに独立したオブジェクト・コードが実行される．システムとしては，プロセッサごとに異なるカーネルが協調して動作することになる．

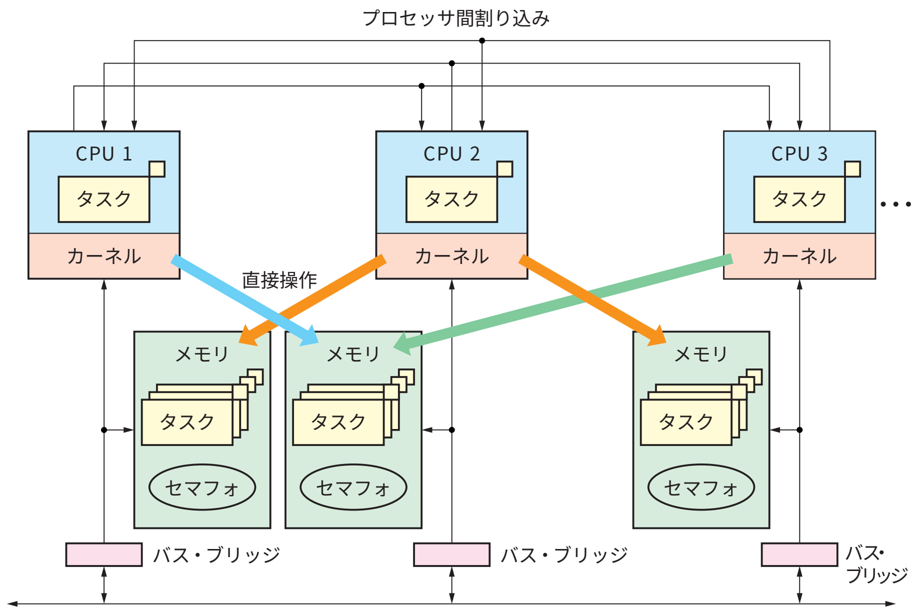


図2
FDMPカーネルの動作イメージ
プロセッサごとに異なるカーネル・オブジェクトが動作している．ほかのプロセッサのオブジェクトを操作する場合，そのオブジェクトが属するプロセッサのローカル・メモリを直接操作する．そして，ディスパッチが必要になった場合は割り込みを発行する．

1) 各プロセッサのデータ(rodata[注3]も含む)を置くメモリがすべてのプロセッサから同一のアドレスでアクセス可能であること
2) 任意のプロセッサで割り込み(プロセッサ間割り込み)を発生できること
3) プロセッサ間で排他制御を行うための機構を持つこと
4) プロセッサ間の排他制御機構を用いて，プロセッサ数×2個のロック(アクセスの制限)を設定できること

1)と**2)**は，直接操作法を用いてほかのプロセッサのオブジェクトを操作するために必要な条件です．**3)**の排他制御のための機構とは，例えばプロセッサの持つTAS(test and set)命令[注4]や，ミューテックス(Mutex)回路[注5]を指します．**4)**については，システムに二つのロックで実現可能ですが(詳細は後述)，この条件によりリアルタイム性は低下します．

本カーネルは，現時点(2005年5月)で以下の三つのプロセッサに対応しています．

- 東芝の「MeP(Media Embedded Processor)」
- 米国Altera社の「Nios II(FPGA向けソフト・マクロ)」
- 米国Xilinx社の「MicroBlaze(FPGA向けソフト・マクロ)」

なお，Nios IIとMicroBlazeでFDMPカーネルを動作させる場合，それぞれの標準構成に含まれていないハードウェアを追加する必要があります．

● オブジェクト操作に必要な情報を管理するCCB

FDMPカーネルでは，直接操作法によってほかのプロセッサのオブジェクトを操作するため，ほかのプロセッサに属するオブジェクトの管理ブロック，例えばタスクを管理するためのタスク・コントロール・ブロック(TCB)がどこにあるのかを，プロセッサ間でお互いに知っておく必要があります．そのため，各プロセッサに所属するオブジェクトを操作するために必要なデータ構造へのポインタを，クラス・コントロール・ブロック(CCB)と呼ばれる構造体に登録します．

オブジェクトの操作には，管理ブロック以外にそのオブジェクトのIDの最大値(オブジェクトの最大数)が必要となります．そこで，管理ブロックとIDの最大値を登録するCBTCB(Control Block Table Control Block)という構造体を用意します．CCBにはオブジェクトごとのCBTCBを登録します(**リスト1**)．このほか，CCBには**図3**に示す情報へのポインタが登録されています．

CCBはクラス(プロセッサ)ごとに作成され，これを起動時にほかのプロセッサと交換します．CCBの内容はリン

注3：rodataとは，C言語プログラムでconst指定された値が固定されている変数を配置するメモリ領域のこと．通常はROMに置く．
注4：対象のメモリ領域を読んで，その内容をレジスタもしくはステータス・レジスタに入れる命令．読んだ値が'0'なら対象となるメモリ領域に'1'をセットする．この命令を発行している間，CPUはバスを占有する．そのため，マルチプロセッサでの排他制御に用いられる．
注5：排他制御を行うための回路．TAS命令を持たないプロセッサの場合に用いる．

3



リスト1　CBTCBの内容

```
typedef struct cb_table_control_block {
    void *cb_table; /* 管理ブロックへのポインタ */
    ID   max_id;    /* オブジェクトの最大数 */
} CBTCB;
```

- プロセッサID
- ロック変数
- レディ・キュー関連(redy_queue, redy_primap)
- 実行中のタスクのTCBへのポインタ(runtsk)
- 最高優先順位のタスクのTCBへのポインタ(schedtsk)
- ディスパッチ禁止状態(enadsp)
- タイム・イベント・ヒープ関連(tmevt_heap, last_index)
- タスク起動ルーチンのアドレス(activate_r)
- 周期ハンドラ起動ルーチンのアドレス(call_cychdr)
- 各オブジェクトのCBTCB

図3　CCBに登録される情報
各プロセッサに所属するオブジェクトを操作するために必要なデータ構造へのポインタは，CCBに登録される．

ク情報をもとに静的に作成することも可能ですが，現在のところこの方法には対応していません．

カーネルの中でオブジェクトの管理ブロックを操作する部分では，オブジェクトのIDからクラスIDを取り出して，対応するCCBの登録情報をもとに操作を行います．**リスト2**に，タスクを起動するact_tskについて，シングル・プロセッサ用のJSPカーネルのコードとFDMPカーネルのコードを示します．なお，**リスト2**は実際のコードからチェック・ルーチンやログ・ルーチンを外して，マクロやインライン関数を展開しています．JSPカーネルのact_tskでは，タスクID(tskid)からTCBを取り出します．タスクの起動処理(make_active(tcb))が必要な場合，tcbを引き数にしてその処理を呼び出します．

FDMPカーネルのコードの流れは，基本的にはJSPカーネルのコードと同じです．まず，タスクID(tskid)を16

リスト2　JSPカーネルのact_tskとFDMPカーネルのact_tsk

```
SYSCALL ER
act_tsk(ID tskid)
{
    TCB             *tcb;
    ER              ercd;

    tcb = ((tskid) == TSK_SELF ? runtsk : (&(tcb_table[INDEX_TSK(tskid)])));
    t_lock_cpu();
    if (TSTAT_DORMANT(tcb->tstat)) {
      if (make_active(tcb)) {
          dispatch();
      }
      ercd = E_OK;
    }
    else if (!(tcb->actcnt)) {
      tcb->actcnt = TRUE;
      ercd = E_OK;
    }
    else {
      ercd = E_QOVR;
    }
    t_unlock_cpu();
  exit:
    LOG_ACT_TSK_LEAVE(ercd);
    return(ercd);
}
```

タスクIDからTCBを取り出す

tcbを引き数に，タスク起動処理を呼び出す

（**a**）JSPカーネルのact_tsk

```
SYSCALL ER
act_tsk(ID tskid)
{
    TCB             *tcb;
    ER              ercd;
    CCB             *ccb;
    BOOL      dspreq = FALSE;
    ID        clsid = ((ID)((H)((tskid) >> 16)));

    ccb = ((clsid) == TCLS_SELF ? runccb : (&ccb_table[(clsid) - 1]));
    tskid = ((ID)((H)(tskid)));
    tcb = (tskid) == TSK_SELF ? runtsk : (&(((TCB *)(ccb->tsk.cb_table))
                                          [INDEX_TSK(tskid)]));
  retry:
    T_ACQUIRE_LOCK(ccb->tsk_lock);
    if (TSTAT_DORMANT(tcb->tstat)) {
      if (make_active(ccb, tcb)) {
          dspreq = dispatch_request(ccb->prcid);
      }
      ercd = E_OK;
    }
    else if (!(tcb->actcnt)) {
      tcb->actcnt = TRUE;
      ercd = E_OK;
    }
    else {
      ercd = E_QOVR;
    }
    T_RELEASE_LOCK_AND_DISPATCH(ccb->tsk_lock, dspreq);
  exit:
    return(ercd);
}
```

タスクIDを16ビット右にシフトして，クラスIDを取り出す

IDに対応するクラスのCCBを取得

タスクIDの下位16ビットを取り出す

タスクIDから，TCBを取得

ccbとtbcを引き数に
タスク起動処理を呼び出す

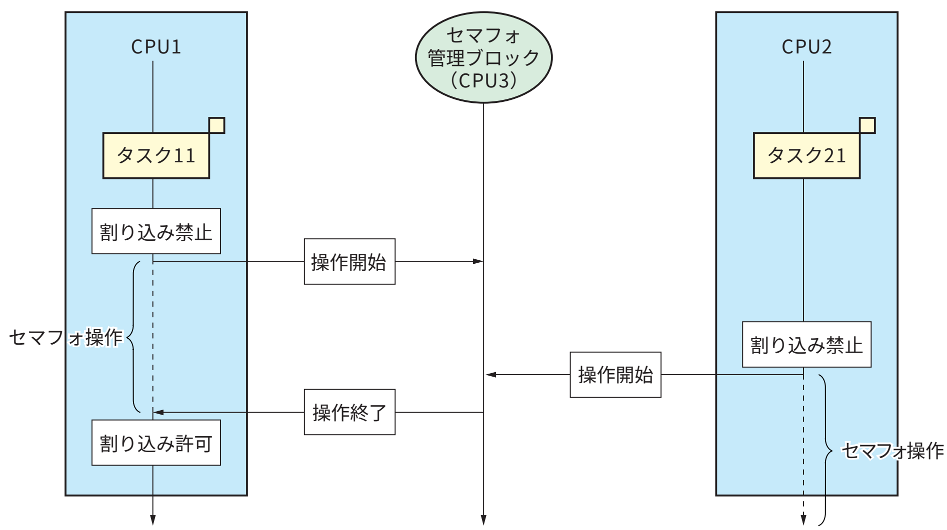
（**b**）FDMPのack_tsk

ビット右にシフトして，クラスID(clsid)を取り出します(**リスト2(b)**)．次に，取り出したクラスIDに対応するクラスのCCB(ccb)を取得します．続いて，タスクIDの下位16ビット(オブジェクトID)を取り出します．対象クラスのCCBに登録されているCBTCB(tsk)のcb_table[注6]を用いてオブジェクトIDからTCBを取得します．その後，タスク起動処理(make_active(ccb, tcb))を呼び出します．FDMPカーネルの場合，tcbだけでなく，ccbも引き数とします．

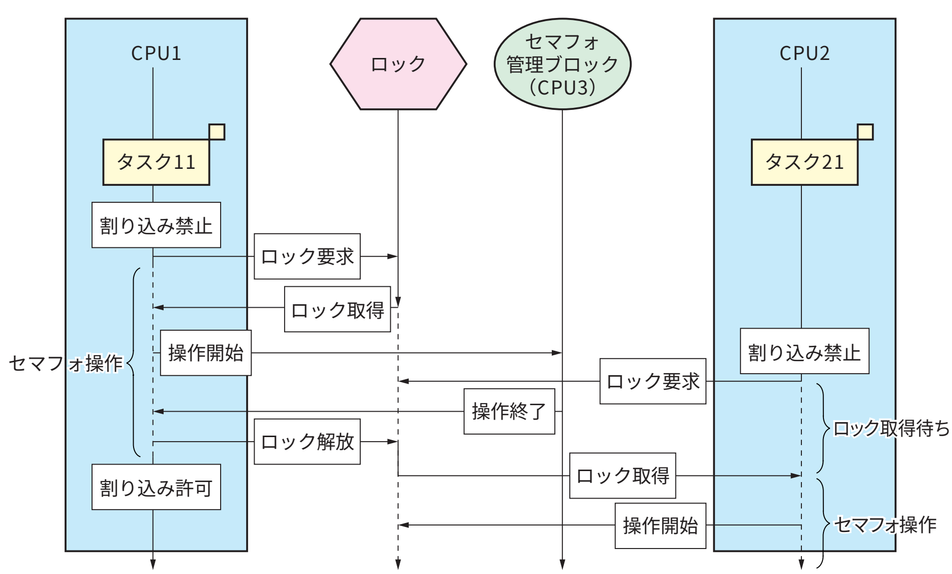
● デッドロックを考慮して二つのロックを用意

FDMPカーネルでは，直接操作法によってほかのプロセッサのオブジェクトを操作します．オブジェクトの管理ブロック，すなわちカーネル内の情報は不可分なく操作しなければならないため，排他制御が必要となります．シングル・プロセッサのカーネルでは，操作の前に割り込みを禁止して，カーネル内の情報の操作中にほかのタスクに切り替わったり，割り込みハンドラが実行されるのを防ぎます．一方，機能分散型マルチプロセッサでは，あるプロセッサで割り込みを禁止したとしても，ほかのプロセッサはその影響を受けないため，割り込みの禁止だけでは排他制御を

注6：CCBにオブジェクトごとに登録するCBTCB構造体のエントリ．そのオブジェクトの管理ブロックへのポインタとなっている．



(a) 割り込みのみで排他制御を実現



(b) セマフォ管理ブロックをガードするロック変数を用意

図4
排他制御
(a)は，割り込みのみで排他制御を実現した場合．プロセッサCPU1のタスク11が割り込みを禁止してプロセッサCPU3のセマフォの管理ブロックを操作する．割り込みを禁止しているため，操作中にCPU1ではタスク11以外の処理単位(タスクや割り込みハンドラ)が実行されることはない．このとき，異なるプロセッサCPU2で動作するタスク21はCPU1と同じようにCPU3のセマフォ管理ブロックの操作を開始することができるため，ほかの方法によるプロセッサ間での排他制御が必要となる．(b)は，セマフォ管理ブロックをガードするロック変数を用意した場合．セマフォ管理ブロックを操作する前にロックを取得する．そのため，CPU1のタスク11がロックを取得してCPU3のセマフォ管理ブロックをアクセスしている間，ほかのプロセッサのタスクがセマフォ操作を行うためにロックの取得を試みるとロックが取得できずに取得待ちとなる．タスク11がセマフォの操作を終えてロックを解放すると，タスク21はロックを取得してセマフォの操作を開始する．



実現できません．そのため，スピン・ロック注7によって複数のプロセッサの間における排他制御を実現しています．すなわち，カーネル内の情報を操作する場合は，割り込みを禁止してプロセッサ内の排他制御を行うとともに，スピン・ロックでロックを取得します（図4）．一つのロックで排他制御を行うカーネル内の情報の単位を「ロック単位」と呼びます．

組み込みシステムで求められるリアルタイム性とプロセッサ数のスケーラビリティを確保するには，ロック単位を適切に設定する必要があります．あまり細かい単位でロックを用意すると，ロック取得のネスト段数が多くなり，処理のオーバヘッドが増大してしまいます．また，少なくとも各プロセッサのロック単位は別々にするべきです．これはあるプロセッサのオブジェクト操作がほかのプロセッサの処理をじゃまするべきではないとう考えからです．

また，できるだけ多くのシステム・コール（タスクからの機能の呼び出し）においてロックの取得順序が一定になるように，ロック単位を決定する必要があります．これにより，デッドロック注8の回避が必要なシステム・コールを最小限にします．例えば，ロックAとロックBの2種類のロック単位があるとします．タスクAがA→Bの順序でロックを取得し，別のタスクBがB→Aの順序でロックを取得すると，デッドロックが発生します（図5）．このため，システム全体でロックの取得順序を一定に定めています．また，定めた順序でロックを取得しないシステム・コールについては，デッドロック回避のためのコードを追加します（下掲のコラム「デッドロック回避」を参照）．

上記のことを踏まえてμITRON仕様のシステム・コールを分析した結果，FDMPカーネルではプロセッサごとにタスク・ロックとオブジェクト・ロックの二つのロック単位を設けることにしました．

- **タスク・ロック**——タスク管理にかかわるデータ構造用のロック．具体的には，TCBの操作の前に取得する．また，時間管理関係の内部データ（タイム・イベント・ヒープ）もタスク・ロックにより排他制御する．
- **オブジェクト・ロック**——同期・通信オブジェクトにかかわるデータ構造用のロック．セマフォやイベント・フラグ，データ・キューのコントロール・ブロックの操作の前に取得する．

また，一般的なシステム・コールはオブジェクト・ロック→タスク・ロックの順でロックを取得するようにします．

ここで，セマフォの取得と解放を例に，ロックの使いかたを説明します（図6）．

プロセッサCPU1に属するタスク1，プロセッサCPU2に属するタスク2，プロセッサCPU3に属するセマフォがあるとします．セマフォはタスク1によってすでに取得されています（資源数は0となる）．タスク2がそのセマフォの取得を試みて（`wai_sem()`），取得待ち状態となります．その後，タスク1がセマフォを返却します（`sig_sem()`）．こ

注7：条件成立までループして待機する方法．ビジー・ウェイトとも言う．
注8：複数のタスクがお互いに資源の解放を待ち合い，処理が停止すること．

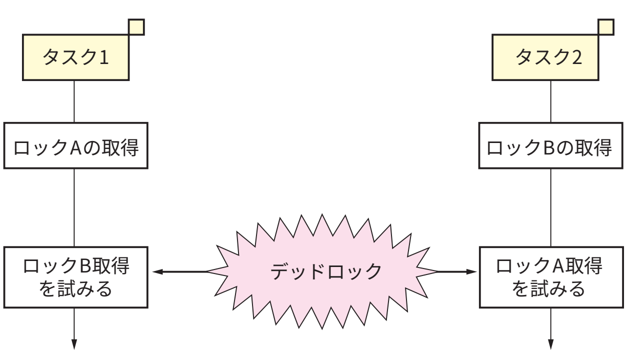


図5　デッドロック

二つのタスクが二つのロックA，Bを取得しようとすると，デッドロックが発生する．ロック取得の順序を一定になるように，コード変更することで回避する．

Column　デッドロック回避

本文で説明したように，多くのシステム・コールでは，オブジェクト・ロック→タスク・ロックの順でロックを取得します．以下に示すいくつかのシステム・コールは，そのままではタスク・ロック→オブジェクト・ロックと逆順にロックを取得するため，デッドロック回避のメカニズムが必要となります．

- `rel_wai()`：待ち状態の強制解除
- `chg_pri()`：タスク優先度の変更
- `ter_tsk()`：タスクの強制終了
- タイムアウト関連のコード

これらのシステム・コールはタスク・ロックを取得してTCBにアクセスした後で初めてロックするべきオブジェクト・ロックがわかります．そのため，一度，タスク・ロックを取得した後，ロックするべきオブジェクト・ロックがわかった時点でいったんタスク・ロックを解放して，あらためてオブジェクト・ロック→タスク・ロックの順序でロックを取得するようにします．

のときのロックの取得・解放は次のとおりです．

●タスク2のwai_sem()の働き

セマフォはCPU3に属するので，タスク2はCPU3のオブジェクト・ロックを取得してセマフォのコントロール・ブロックにアクセスします．アクセスの結果，資源数が0であることがわかり，タスク2はセマフォ獲得待ち状態となります．そのため，自タスク，すなわちタスク2のTCBを操作するため，CPU2のタスク・ロックを取得して待ち状態へ移行します．最後に，二つのロックを手放してディスパッチします．

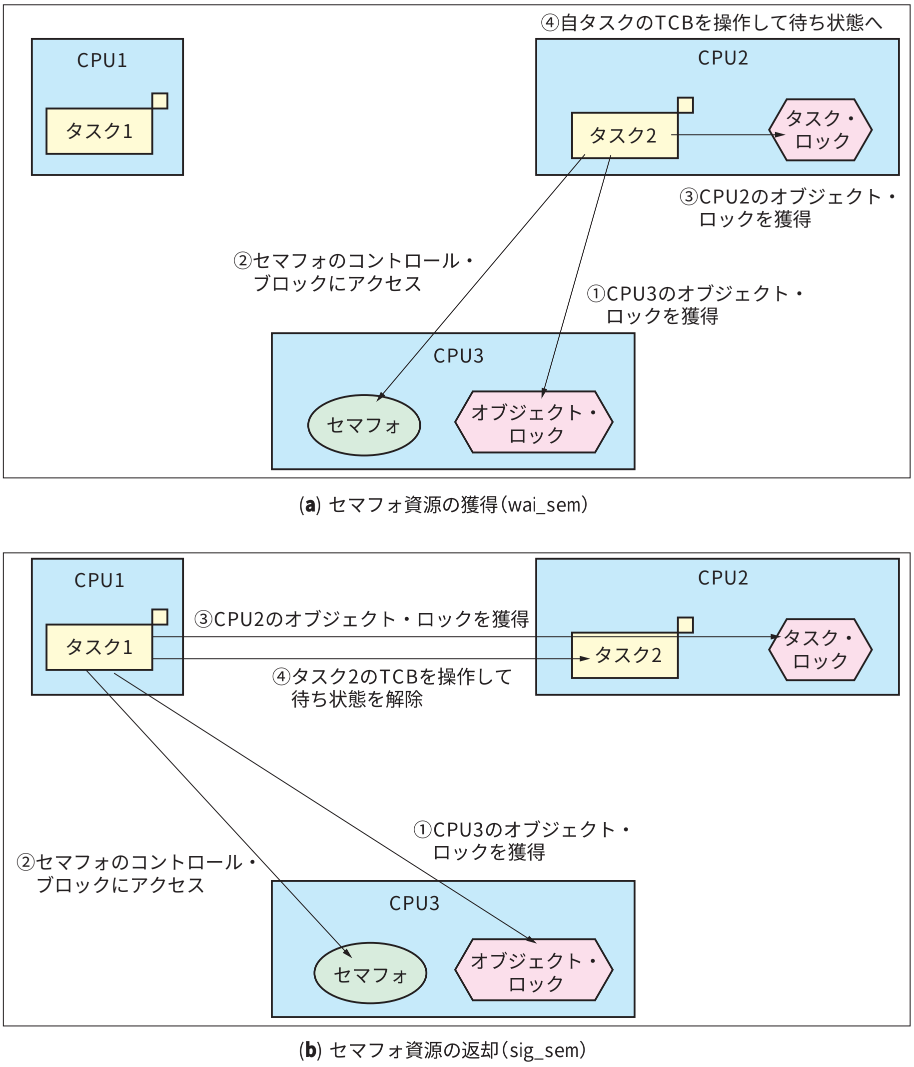
●タスク1のsig_sem()の働き

タスク2と同じように，タスク1はCPU3のオブジェクト・ロックを取得してセマフォのコントロール・ブロックにアクセスします．その結果，タスク2が待ち状態になっていることがわかるので，CPU2のタスク・ロックを取得してタスク2を待ち状態から解放します．そして，CPU2でディスパッチが必要になった場合(タスク2が実行可能状態のタスクの中でもっとも優先度が高い場合)，CPU2に対してプロセッサ間の割り込みを発行してディスパッチを要求します．最後に，取得した二つのロックを解放します．

● ロックの取得と割り込み発生のタイミングは重要

システム・コールの実行には，プロセッサ間の排他制御とプロセッサ内の排他制御が必要となります．FDMPカーネルでは，前者はロックによって，後者は割り込み禁止によって実現しています．この二つの排他制御の関係は難しく，適切に扱わなければなりません．

例えば，ロックを取得して割り込みを禁止する前に割り込みが入ってしまうと，それを受け付けてしまい，割り込み処理の間はほかのプロセッサを待たせてしまうことにな



(a) セマフォ資源の獲得(wai_sem)

(b) セマフォ資源の返却(sig_sem)

図6
ロックの使いかた

(a)では，プロセッサCPU2に属するタスク2が別のプロセッサCPU3に属するセマフォ資源の獲得を，wai_sem()を用いて試みている．その結果，資源数が0であることがわかり，自タスクを待ち状態に移行させる．(b)はプロセッサCPU1に属するタスク1がプロセッサCPU3に属するセマフォ資源をsig_sem()により返却するようすを示している．タスク2が待ち状態になっていることがわかるので，その待ち状態を解除する．

3



ります．逆に，割り込みを禁止してロックの取得を試みると，割り込み禁止時間が長くなってしまいます．理想的な方法は，ロックと割り込みの二つを同時に行うことです．

この点について，現在のFDMPカーネルは理想的な解決法に至っておらず，今後の課題となっています．FDMPカーネルの現在の実装の例を，セマフォの獲得（`sig_sem()`）

リスト3
セマフォの獲得

```
SYSCALL ER
sig_sem(ID semid)
{
    SEMCB   *semcb;
    TCB     *tcb;
    ER      ercd;
    CCB     *ccb, *tccb;
    BOOL    dspreq = FALSE;

    ccb = T_CHECK_CLSID_CCB(semid);
    semid = remove_clsid(semid);
    semcb = get_semcb(ccb, semid);

  retry:
    //割り込み禁止
    t_lock_cpu();
    //オブジェクト・ロックの取得
    if ((ccb->obj_lock).lock_ops.acquire_lock(&(tccb->tsk_lock))) {
          //割り込み許可
          t_unlock_cpu();
          //ここで割り込みを受け付ける
          goto retry;
    }
    if (!(queue_empty(&(semcb->wait_queue)))) {   ┐
      tcb = (TCB *)(semcb->wait_queue.next);      ├ 非破壊コードでなければならない
      tccb = get_ccb(tcb->clsid);                 ┘
      //タスク・ロックの取得
      if ((tccb->tsk_lock).lock_ops.acquire_lock(&(tccb->tsk_lock))) {
          //ロックが取得できなかった場合，オブジェクト・ロックを解放
          (ccb->obj_lock).lock_ops.release_lock(&(ccb->obj_lock));
          //割り込み許可
          t_unlock_cpu();
          //ここで割り込みを受け付ける
          goto retry;
      }
      /* 待ちキューから削除する */
      queue_delete((QUEUE *) tcb);
      if (wait_complete(tccb, tcb)) {
          dspreq = dispatch_request(tccb->prcid);
      }
      //タスク・ロックの解放
      (tccb->tsk_lock).lock_ops.release_lock(&(tccb->tsk_lock));
      ercd = E_OK;
    }
    else if (semcb->semcnt < semcb->seminib->maxsem) {
      semcb->semcnt += 1;
      ercd = E_OK;
    }
    else {
      ercd = E_QOVR;
    }
    //オブジェクト・ロックの解放
    (ccb->obj_lock).lock_ops.release_lock(&(ccb->obj_lock));
    if (dspreq) {
      dispatch();
    }
    t_unlock_cpu();

  exit:
    LOG_SIG_SEM_LEAVE(ercd);
    return(ercd);
}
BOOL
acquire_lock(LOCKOBJ *lockobj)
{
    do {
      //ロック取得
      if (mutex_try_lock((unsigned int)(lockobj->lock_ptr), PRCID) != 0){
          //ロックを取得できた
          return (FALSE);
      }
    } while (current_ipending() == 0); //割り込みが入っているかチェック
    return (TRUE);
}
```

を用いて解説します(**リスト3**).

1) まず，割り込みを禁止してプロセッサ内の排他制御を実現する
2) オブジェクト・ロックを取得する．ロックの取得を行う `acquire_lock()` では，スピン・ロックでロックの取得を試みる．ロックの取得の施行ごとに割り込み要求があるかどうかをチェックする．ロックが取得できればFALSEを返し，ロックを取得できずに割り込みが入ればTRUEを返す
3) ロックが取得できなかった場合は割り込みを許可して，1)の割り込み禁止以降を繰り返す．なお，割り込み要求があるため，割り込みを許可した時点で割り込みを受け付ける
4) セマフォ管理ブロックへのアクセスの結果，タスク・ロックの取得が必要となった場合(待ちタスクが存在する)，そのタスクの属するプロセッサのCCBを取り出してタスク・ロックの取得を試みる
5) タスク・ロックが取得できなかった場合，すなわちロックの取得の前に割り込み要求があった場合はオブジェクト・ロックを解放して1)に戻る．3)と同じように割り込み要求があるため，割り込みを許可した時点で割り込みが受け付けられる

このように，タスク・ロックを取得できない場合，オブジェクト・ロックの取得以降を繰り返します．このため，オブジェクト・ロックの取得からタスク・ロックの取得の間のコードは繰り返し実行される可能性があるので，非破壊コードでなければなりません．

2 FPGAのソフト・マクロCPUに実装

ここで，FDMPカーネルの実装例を紹介します．今回は，Altera社のFPGA用ソフト・マクロCPU「Nios II」とXilinx社のFPGA用ソフト・マクロCPU「MicroBlaze」をターゲットとしました(右掲のコラム「ソフト・マクロCPUの使いごこち」を参照).

注9：タスクを休止状態から実行可能状態に移行させる際に呼び出されるシステム依存部の関数(カーネル内のコード).
注10：割り込みハンドラからの復帰時に呼び出されるカーネル内のコード.
注11：CPU例外ハンドラからの復帰時に呼び出されるカーネル内のコード.

● ターゲット依存部はJSPカーネルとほとんど同じ

前述のようにFDMPカーネルはJSPカーネルをベースとしています．JSPカーネルはターゲット・システム(プロセッサやボード)に依存した部分(ターゲット依存部)と依存しない部分に大きく分かれています．

新しいターゲットにFDMPカーネルを移植する場合は，このターゲット依存部を開発する必要があります．ただし，ターゲット依存部はほぼJSPカーネルのものと同じであるため，JSPカーネルが動作していれば移植は容易です．筆者の場合，JSPカーネルが動作していたNios IIへの移植は，ハードウェアの開発も含めて1日でできました．JSPカーネルのプロセッサ依存部とFDMPカーネルの相違点は次のとおりです．

- ロック用関数の追加
- 起動時のプロセッサ間の同期用関数の追加
- プロセッサ間の割り込み関数の追加
- いくつかの関数の修正．例えば，activate_context()注9，ret_int注10，ret_exc注11など

● Nios IIへの移植――割り込み回路は新たに開発

Nios IIを用いたシステムは，Altera社のFPGA用開発

3

Column ソフト・マクロCPUの使いごこち

Nios IIやMicroBalzeなど，FPGA向けのソフト・マクロのプロセッサは，プロセッサ・コアの機能や仕様がバージョンアップとともに変わっていきます．改善や改良そのものは問題ないと思うのですが，前のバージョンとどの部分が変わったのか，その差分をマニュアルなどで明確に紹介していないため，バージョンアップのたびにマニュアルを端から端までチェックしなければなりません．

また，プロセッサ・コアの変更とともに開発環境，とくにコンパイラのバージョンアップが必要となります．そもそも新しいプロセッサのコンパイラは安定していないうえに，機能や仕様の変更によってよく不ぐあいも発生します．執筆当時，Nios IIとMicroBlazeのコンパイラ(gcc)の最新バージョンはそれぞれ,「gcc version 3.4.1 (Nios II 1.1付属)」と「gcc version 2.95.3-4 (EDK 6.3付属)」でしたが，FDMPカーネルは動作しませんでした．筆者は，その一つ前のバージョンである「gcc version 3.3.3 (Nios II 1.0付属)」,「gcc version 2.95.3-4 (EDK 6.2.1付属)」を使用しています(なお，現在，それぞれの最新のバージョン「Nios II 5.0」,「EDK 7.0」でコンパイラがバージョンアップしている).

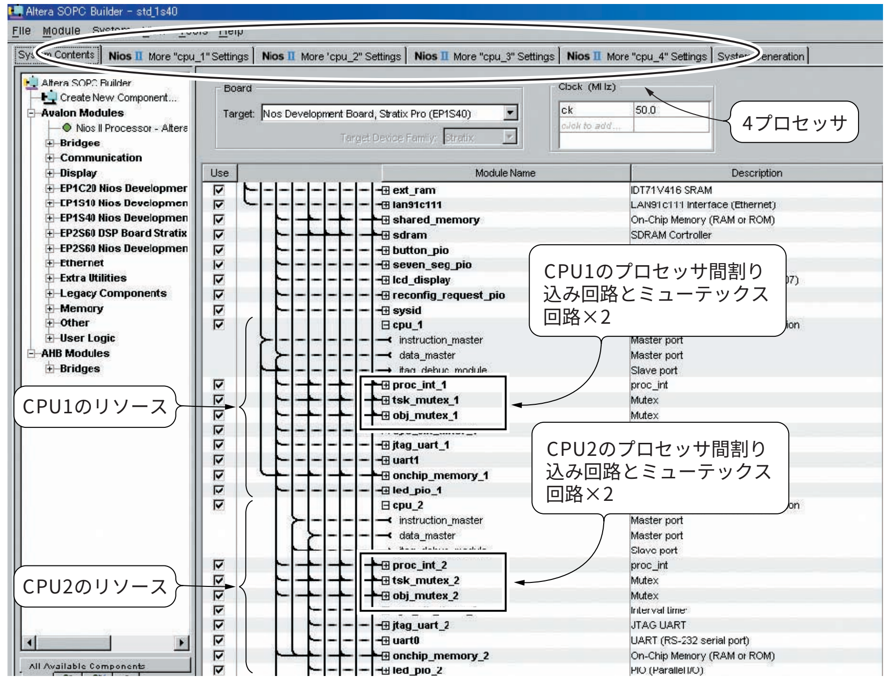


図7
SOPC Builderで作成したNios IIの4プロセッサ・システム

FDMPカーネルを動作させている4個のNios II CPUコアのシステムをSOPC Builderで表示した．プロセッサごとに割り込み回路（proc_int）とオブジェクト・ロック/タスク・ロック用のミューテックス回路（mutex）を持つ．

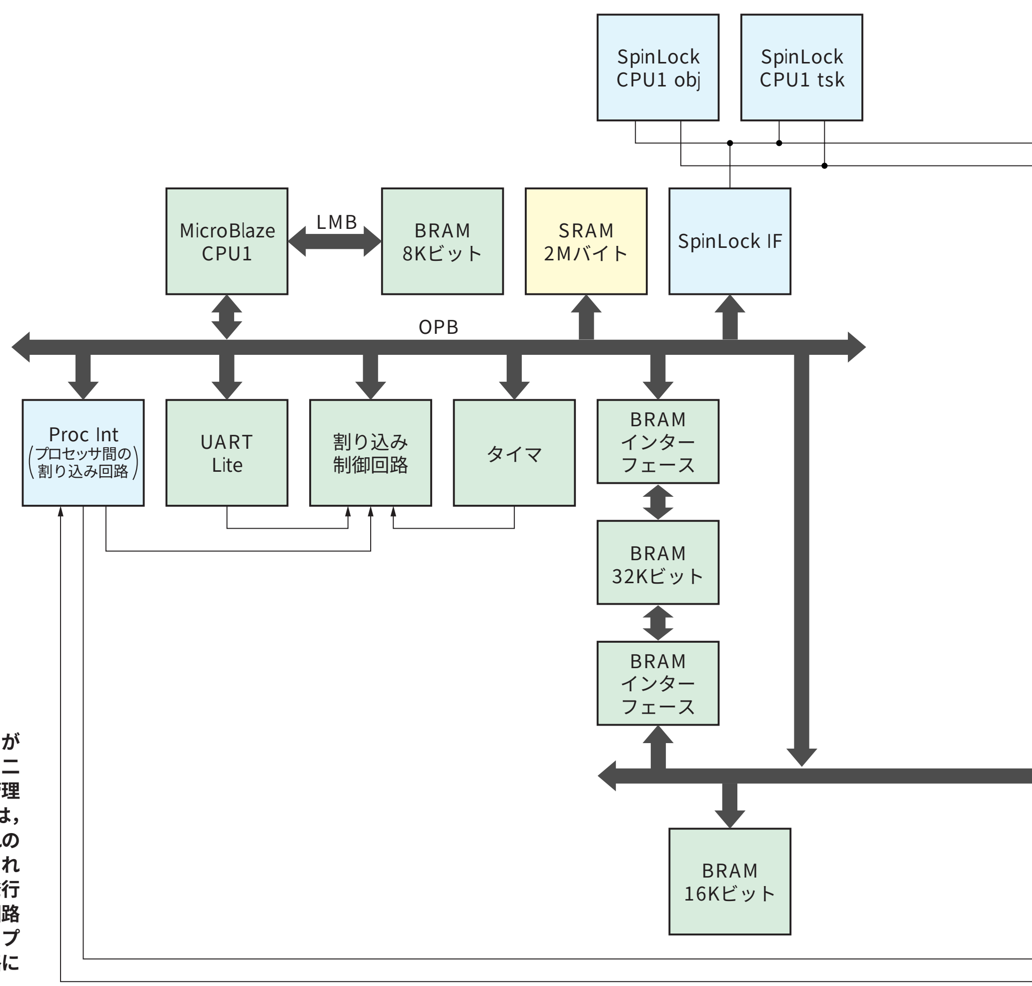


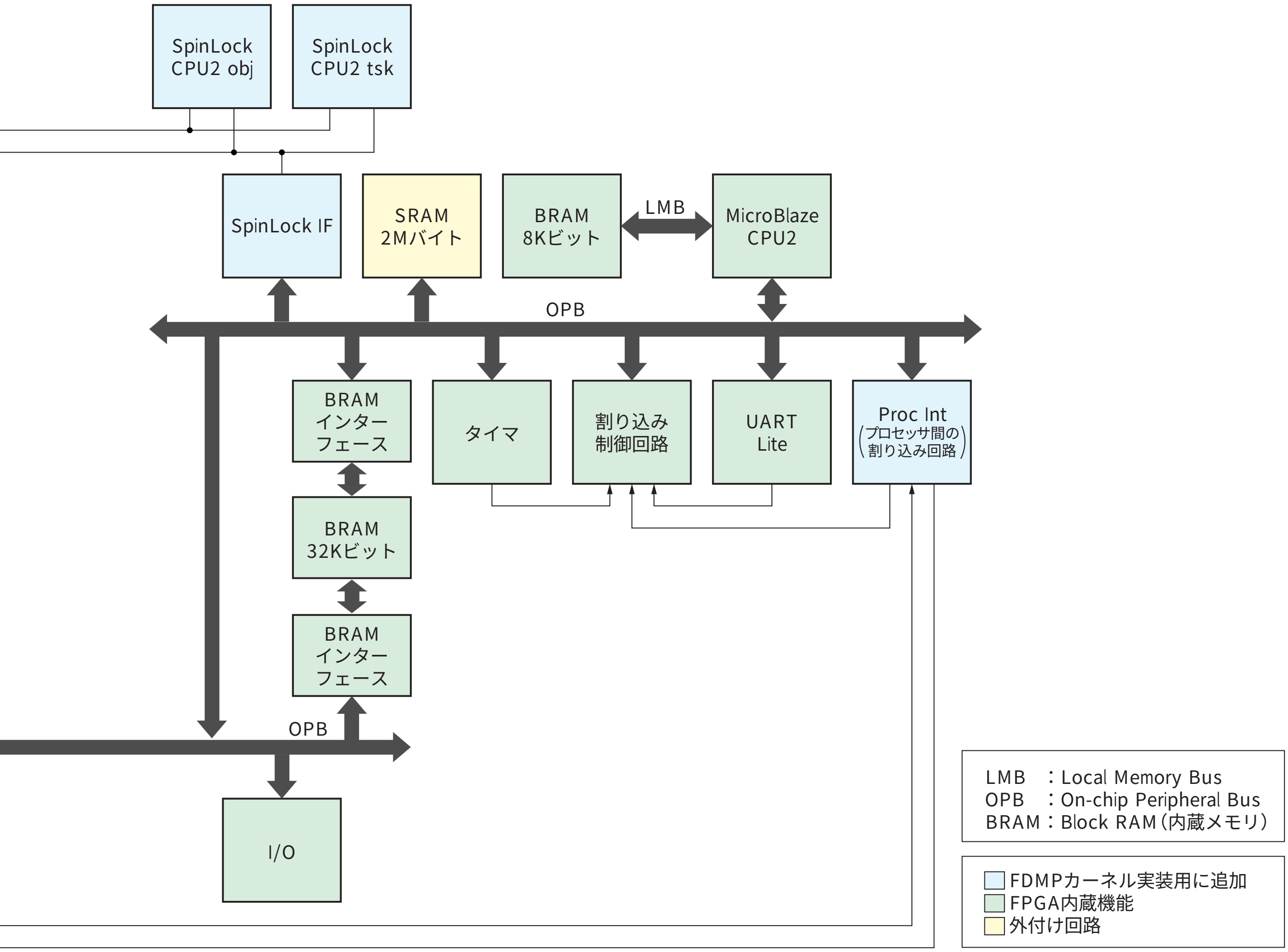
図9
MicroBlazeを2個使った場合のシステム構成図

MicroBlazeのバス・ブリッジの仕様上，デッドロックが避けられない．そのため，デュアルポート・メモリを二つのバスに接続して，このメモリにオブジェクトの管理ブロックを置いている．また，プロセッサ間割り込みは，Proc Int回路によって発生させる．この回路はそれぞれのプロセッサのローカル・バスにあり，お互いに結線されている．そして，任意の接続先に対して割り込みを発行できる．排他制御は，SpinLock IF回路とSpinLock回路の組み合わせで実現する．それぞれのロックに対して，プロセッサはSpinLock IF回路を経由してSpinLock回路にアクセスする．



図8　図7のシステムの動作確認

4個のCPUコアでそれぞれFDMPカーネルが起動している．起動してバナー（起動時にプロセッサ名やコピーライトを表示する部分）を出力した後，プロセッサ間で同期をとり，CCBの情報を交換している（waiting for EXCCB initializedの部分）．



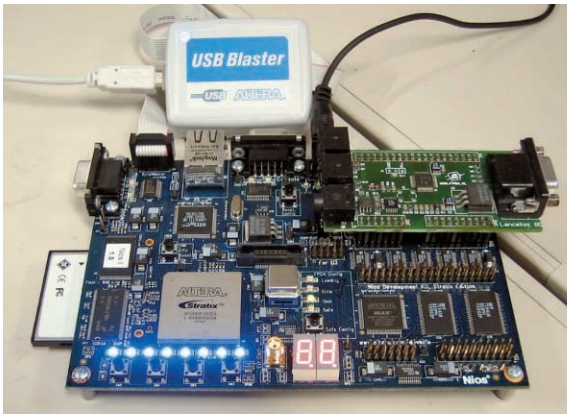


写真1　Nios II用評価ボード
FPGA内にNios IIプロセッサを4セット実装して，FDMPカーネルを動作させている．

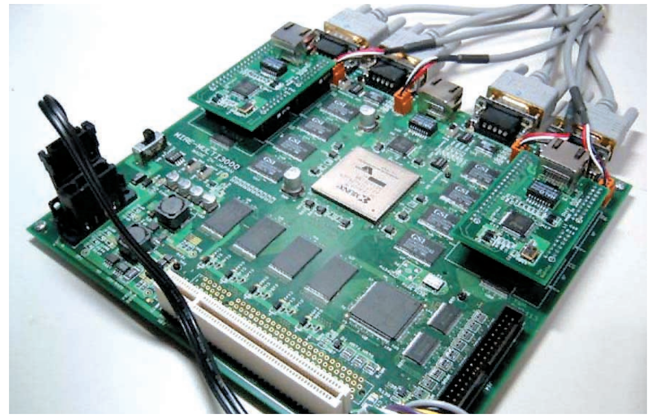


写真2　MicroBlaze用評価ボード
機能分散型マルチプロセッサOSの研究・開発のために作製したボード．基板上には，4プロセッサ分のリソース（フラッシュ・メモリ，SRAM，UART）を実装している．

環境「Quartus II」に含まれる「SOPC Builder」を用いて設計します．Nios II版のFDMPカーネルでは，ミューテックス回路を用いてロックを実現しています．また，SOPC Builderに含まれるNios II用の標準ペリフェラルにはプロセッサ間の割り込みを発生させる回路が含まれていないため，プロセッサ間の割り込み回路を新たに開発しました（図7，図8）．

筆者は，開発用としてAltera社のNios II開発キット「Stratixプロフェッショナル・エディション」を用いて，4個のプロセッサ・コアが動作するシステムを構築しています（写真1）．Nios IIでは，FPGAのコンフィグレーション用のUSB Blasterを用いて，複数のコンソールとプロセッサごとのデバッガ，およびデバッグ・ユニットの接続を1本のUSBケーブルで実現できました．

● MicroBlazeへの移植――バス・アクセスをくふう

Nios IIのSOPC Builderと同等の機能を持つ開発ツールとして，MicroBlazeでは「Platform Studio」が用意されています．MicroBlazeでは，FDMPカーネルを動作させるためにプロセッサ間の割り込み回路とミューテックス回路を開発しました．さらに，開発用ボードとして1個のFPGAの周りに4セット分のプロセッサ・リソース（SRAM，二つのUART）を実装したボードを開発しました（写真2）．

MicroBlazeはコンソールごとにシリアル・ポートが必要となります．MicroBlazeを用いたシステムでは，バス・ブリッジを用いてプロセッサごとのバスをグローバル・バスと接続しますが，このバス・ブリッジは簡易的なものであり，バス・アクセスの際にデッドロックが発生する可能性があります（図9）．そこで，各プロセッサのメモリはデュアルポート・メモリとし，グローバル・バスとローカル・バスの両方に接続することで，デッドロックを回避する必要があります．

なお，ここで紹介したNios IIやMicroBlazeに追加したハードウェアのソース・コードは，今後，http://www.ertl.jp/~honda/FDMP/にて公開する予定です（現在，準備中）．

＊　＊　＊

2005年5月下旬にTOPPERSプロジェクトから同プロジェクトの会員向けに，早期リリースとしてTOPPERS/FDMPカーネルのソース・コードを公開しました．今後，この早期リリースの結果をフィードバックして，遅くとも年内には一般向けに公開する予定です．

ほんだ・しんや
名古屋大学 情報連携基盤センター 名古屋大学組込みソフトウェア人材養成プログラム

<筆者プロフィール>
本田晋也．長かった学生生活を終えて，今年から名古屋大学組込みソフトウエア技術者人材養成プログラム（NEXCESS）で研究員をやっています．最近，周りから「スーパーなんとかですね」と言われて気恥ずかしいです．